



Hotpatching the Hotpatcher

Stealth File-less DLL Injection

Alex Ionescu, Chief Architect
SyScan 2013

@aionescu
alex@crowdstrike.com

Bio

- Reverse engineered Windows kernel since 1999
 - Previously lead kernel developer for ReactOS Project
- Interned at Apple for a few years (Core Platform Team)
- Co-author of Windows Internals 5th and 6th Edition
 - Also instructor and contributor to Windows Internals seminar for David Solomon Expert seminars
- Founded Winsider Seminars & Solutions Inc., to provide services and Windows Internals training for enterprise/government
- Now Chief Architect at CrowdStrike
 - Security startup focused on attribution, received \$26M in funding

Introduction

Outline

- Introduction
- Motivation
 - Previous Work
- Baby Steps
 - The Hotpatch API and DLL Injection
- Bigger Leaps
 - Known DLL Section Object Misdirection and Hijacking
- Moonshot
 - The Hotpatch CodeInfo API – how to write a real patch
 - How to fail at getting anywhere with it
 - Based Sections to the rescue
- Defense

What This Talk Is (Not) About

- A rootkit/malware persistence technique
 - Hidden from the usual HIPS/AV callbacks and mechanisms
 - Usable from kernel *and* user
- **Require privileged user token -- not a 'security vulnerability'**

Motivation

- We want to load code in a stealthy way inside another process
- Or, we are in kernel (first-stage/early payload) and want to go into user-mode (second-stage/persistent payload)
- Most of these techniques require
 - Writing a file to disk (visible)
 - Injecting a DLL (visible)
- Current stealthy techniques don't have this problem, but
 - Implement their own loader code (complex, especially in kernel use case)
 - Can't hope to fully implement all loader intricacies – final payload is simple
 - TLS Initializers? AMD64 Relocations? Shim Engine Callbacks? SafeSEH Initialization?
- Can we get the benefits of a custom loader...
 - Without having to write one?

Previous Work

- EliCZ “*Microsoft Patching Internals*” on OpenRCE
 - Most commonly referenced source of information/header files
 - Used in, among other things, CanSecWest ASLR/ROP bypass talk
 - However, changes were done in Vista/Windows 7 to many of the structures
 - No public officially updated header
- Derek Soeder “*PiXiE: A Self-Propagating Network Boot Virus for Windows*” on eEye
 - Used Known DLL Section Object Hijacking for user-mode payload
 - But required patching the kernel (not usable from user-mode)
- Johannes Passing (blog)
 - Wrote pseudo-code for Vista+ Hotpatch API
 - Mainly focused on kernel driver patching, not user-mode

Baby Steps

What is Hotpatching?

- First introduced in Windows XP SP2
 - Incomplete implementation, was never used
- Officially supported and implemented in Windows Server 2003
 - Hopatching Beta program opened for testing
 - List of KB's is on Elicz's paper and Johannes' blog
- Added code to compiler and linker (/HOTPADMIN, /HOTPATCH)
 - Adds 5 “nop” instructions on top of every function (before the entry)
 - Used to store absolute call address
 - Adds “mov edi, edi” instruction at the start of every function (at the entry)
 - Used to store 2-byte relative jump to the absolute call
- Less than a dozen Hotpatch updates ever released
 - Project was eventually abandoned
 - Strangely, Vista improved & updated the Hotpatch API

Introduction to the Hotpatch API

- Available through *NtSetSystemInformation*
 - With class *SystemHotpatchInformation* and structure `SYSTEM_HOTPATCH_CODE_INFORMATION`
- Different modes supported
 - `FLG_HOTPATCH_KERNEL`: Driver Hotpatch
 - `FLG_HOTPATCH_INJECT`: User Hotpatch
 - `FLG_HOTPATCH_NAME_INFO`: Required for Target/Patch Name
 - `FLG_HOTPATCH_MAP_ATOMIC_SWAP`: Section Object Swapping
 - `FLG_HOTPATCH_RELOAD_NTDLL`: NTDLL Section Object Refresh
- Patch can either be applied, or removed:
 - `FLT_HOTPATCH_ACTIVE`

DLL Injection with the Hotpatch API

- Callable from kernel as well as from user-mode
- `FLG_HOTPATCH_INJECT` allows the kernel to perform thread injection
 - Most AV/HIPS systems do not flag a remote thread created by the System process
 - `ntdll!LdrHotpatchRoutine` takes control
- `FLG_HOTPATCH_NAME_INFO` instructs the thread to load the respective hotpatch DLL
 - The “Target” DLL must exist
 - Simply pick “`ntdll.dll`” as the target – guaranteed to exist
- Hotpatch DLL is loaded with *LdrLoadDll* (native *LoadLibrary*)
 - **`LOAD_LIBRARY_AS_DATAFILE` is not used!**
 - Full API imports, relocation, and DllMain processing is performed

Surviving as an Injected DLL

- *DllMain* is the only piece of code that will run while in the hotpatching process
- After it exits, the hotpatcher will try to find valid hotpatch data (more on this later)
 - If no data is found, the thread is terminated and the DLL unloaded!
- One should therefore:
 - Use *CreateThread* and/or the Vista Thread Pool API to spawn off a new thread
 - Use *LdrAddRefDll*
 - Makes the DLL immune to the hotpatcher's *LdrUnloadDll*
- ~15 lines of code and you've got a universally loadable DLL
 - But it has to be on disk...

Bigger Leaps

Known DLLs

- Caching mechanism in Windows that speeds up application load time
- SMSS (Session Manager) reads registry array:
 - HKLM\System\CurrentControlSet\Control\Session Manager\KnownDLLs
 - All must be under HKLM\System\CCS\Control\KnownDLLs\DllDirectory
- Each DLL is mapped with the section API, but also given a name
 - Imagine *CreateFileMapping* with a filename AND an object name
 - Name is in the \KnownDlls object directory
 - \KnownDlls32 and related registry key used for WOW64 support
- Also prevents someone from overwriting a System DLL and having that new DLL be in use
 - This is why if you rename kernel32.dll to kernel33.dll, and drop in a new kernel32.dll, further DLL loads will **still** see the old kernel32.dll!

Known DLL Section Object Misdirection

- Although new known DLLs are supposed to be added only through the registry (and must be in System32)...
 - One can manually create a new section object in \KnownDlls
- The loader will always check \KnownDlls\foo.dll first, before trying to access the disk
 - Loader does not check if the DLL is “supposed” to be a known DLL . It trusts that if it’s there, there must be a reason
- Recall that the Hotpatch API has a special “section object swap” flag
 - In fact, it is used for this exact reason: when a hotpatch updates a known DLL on disk, the patch would be ignored due to the \KnownDll mechanism
- One can read & map evil.dll into \KnownDlls\evil.dll
 - And then use the hotpatch API to swap with \KnownDlls\kernel32.dll

Known DLL Section Object Hijacking

- The loader has an interesting “bug”/behavior:
 - It trusts that the `\KnownDll` section object is a `SEC_IMAGE` section
 - i.e.: That it really is an image file on disk
- In other words, the manually created `\KnownDll` object does not have to be a DLL on disk...
 - It can be any random piece of shared data created through *NtCreateSection/CreateFileMapping*
- The loader does use *NtQuerySection* to get information on the section however...
 - And this API fails for non-image sections
 - But the loader ignores the failure code!
- So can we load arbitrary “DLLs” that don’t really exist on disk?

Known DLL Section Object Hijacking Advantages

- Ultimately, when the loader does *NtMapViewOfSection*, this goes inside the kernel
- Section mappings notifications are delivered to
 - User-mode debugger (to load symbols and be notified of the new DLL)
 - Kernel-mode debugger (same reason)
 - AV and other drivers (*PsSetImageLoadNotifyRoutine*)
- ...but only for SEC_IMAGE mappings
 - Otherwise, every single random piece of shared data would cause a notify
- Loading a *data* section object through the loader results in absolutely no kernel/debugger notification that a DLL was loaded

But there is a second “bug”...

- When the loader calls *NtMapViewOfSection*, it passes in `PAGE_READWRITE` as the protection mask!
 - For `SEC_IMAGE` sections, the protection mask is ignored, however
 - Real protection comes from the PE section information
- In our case, we *don't* have a `SEC_IMAGE` mapping...
 - So the entire “DLL” gets mapped without execute rights
 - The talk would be over if this we were in 2001 with Windows XP < SP2...
 - Even making/forcing the section object to have `PAGE_EXECUTE_READWRITE` doesn't help – kernel always obeys caller
- Derek Soeder got around this in PiXiE by patching the kernel
 - Making it always hand off `PAGE_EXECUTE_READ_WRITE`
 - This was back in 2004, before PatchGuard, and requires a driver to work...

Moonshot

Hotpatching the loader

- What if, instead of patching the kernel memory manager, we could patch the loader itself
 - Make it send `PAGE_EXECUTE_READWRITE` (0x40) instead of `PAGE_READWRITE` (0x4)
- One could do it from the kernel...
 - Requires MDLs, changing protection and other visible system changes
- Could also be done from user-mode
 - Again, starts requiring very noisy APIs to modify the other process' memory
- What if we actually used the hotpatcher itself?
 - Instead of just injecting an evil file-less DLL, inject real hotpatch data
 - Patch would now be done by the *LdrHotpatchRoutine* and appear to be coming from the process itself (no real visibility)

Intro to the Hotpatching CodeInfo API

- When the injected DLL is “valid”, *LdrHotpatchRoutine* performs preparatory work
 - Validations, fixups, etc (TBD)
- Calls *NtSetSystemInformation* one more time with nothing but `FLG_HOTPATCH_ACTIVE`
 - Kernel now reads the “CodeInfo” data from the `SYSTEM_HOTPATCH_CODE_INFORMATION`
 - Creates a locked & probed writable MDL for the user-mode mapping
 - Uses an exported API to send a DPC to all processors
 - Each processor raises to `IPI_LEVEL - 1`
 - All processors synchronize on a barrier such that only one can pass
 - Winning processor does the *RtlCopyMemory* (*memcpy*) call to do the patch

Writing a real patch DLL

- One must first start with HOTPATCH_HEADER
 - Not documented, but leaked in WRK, EliCZ's paper, PDBS...
 - Magic must be set to '1TOH' (HOT1)
 - Version must be set to MAKELONG(1,0) -> 0x10000
- Next step is identifying the module
 - MD5, SHA, PeHash + name of the module to be patched
 - Means you don't even have to hand-write your own validation!
 - In the demo we use HOTP_ID_None and hard-code the address
- Next, the header can have (all must be RVAs)
 - Fixups (when the patch depends on the target's base address)
 - Validations (to validate that each patch matches the expected bytes)
 - Hooks (the actual patches)

Writing a real patch DLL (continued)

- Once the HOTPATCH_HEADER is written, it must be included in a special PE section
 - “.hotp1 ” (note two trailing spaces)
 - Section size must be large enough to hold HOTPATCH_HEADER plus all relative offsets of the header
- Next, write the relocations and validations for each patch location
 - For demo purposes: laziness. No relocations and no validations – assume all will work fine. (Set counts to 0 in the header)
- Finally, write the actual code patches (“hooks”):
 - Different hooks exist (relative jump, absolute call, AMD64 and IA64 branches, etc...)
 - We also get two absolute value options: VA32 and VA64
 - These allow writing any virtual address at any location into the binary

Writing the actual patch

- We are going to identify the 4-byte aligned address (just for safety) of the “push 0x4” instruction that sends `PAGE_READWRITE` to *NtMapViewOfSection*
 - Save its RVA and write it into the `HOTPATCH_HOOK` as a “target RVA”
- Write down the current 4-byte opcode(s), and change 0x4 to 0x40
 - Store this new 4-byte assembly code as a VA32 “patch RVA”.

...and failing at it

- Turns out the hotpatcher validates all patch types:
 - Is a 2-byte relative jump to an RVA that's in the patch module?
 - Is the address to patch an RVA in the target module?
 - etc...
- This check kills us:
 - Is the 32-bit VA (when using VA32) an actual RVA within the patch module?
- Because our resulting "VA" is actual assembly code, something like:
 - `data->HotpatchHook.HotpRva = 0x1075ff40;`
 - And we don't have a DLL with 0x1075ff40 bytes in it...
- But how does the hotpatcher know how big our DLL is?
 - i.e.: Where does it get the size of our DLL?

Tricking the RVA check... and failing again

- It turns out that the “SizeOfImage” field in the PE header is checked.
 - Normally you can’t fake this: setting a huge value would’ve caused the memory manager’s loader to try mapping a huge image
 - But we’re not loaded as a PE file by the memory manager, are we?
- Simple, obvious, dumb fix:
 - Set SizeOfImage to 0xFFFFFFFF (could be smaller, but this covers all cases) in the fake PE header associated with the in-memory hotpatch image
- OK! Now the “hook” passes validation checks, and the kernel proceeds to do the patch!
 - But instead of writing 0x1075ff40, it writes 0x1195ff40, or 0x1077ff40, or 0x1279ff40...

Why are we failing?

- What is happening is that the hotpatcher is “helpfully” taking relocations into account, and realizing that our hotpatch DLL did not load at its expected base address
 - Microsoft’s internal compiler that generates HOTPATCH_HOOKs must be storing the non-relocated RVA (without relocation records)
 - And so the kernel is “fixing it up”.
- If we knew where our hotpatch DLL loads (or made it /FIXED), we could simply set our Rva to KnownBase – DesiredValue
 - But this implies breaking ASLR
 - Or hoping that our /FIXED address will never be in use
- So: is there a way to reliably predict where our DLL will load?
 - Yes! SEC_BASED

SEC_BASED to the Rescue!

- SEC_BASED creates a “based section object”
 - The first time the section object is created, a top-down search of all system-wide based images is made
 - A predictable, high-VA is always picked (i.e.: 0x7F6E0000)
 - Additionally, the *same* VA is always used
- So our first-stage hotpatch injector can simply check the base address of the mapping it’s using to build the hotpatch
 - And use that as the expected “VA”.
- The best part about SEC_BASED...
 - ... is that it fixes another problem we’re about to get into

What's next?

- Now the patch works:
 - *ntdll!LdrpMapViewOfSection* sends `PAGE_EXECUTE_READWRITE (0x40)`
 - Our hotpatch DLL is loaded with fully executable pages
- But DllMain is executed ***before*** the patch
 - So we still crash before any of this!
- Easy fix: set Entrypoint to NULL, so that we only behave as a hotpatch
 - Then, run the Hotpatch API one more time, but THIS time, don't provide any hotpatch data
 - Instead, provide the memory of "real" DLL that should be injected
 - Including the EntryPoint
- Aaaand.....
 - it crashes

SEC_BASED Kicking More Ass

- The crash is depressing
 - The loader does NOT relocate our fake DLL, and so all imports, TLS initialization, SAFESEH capture crashes miserably
- Relocation is based on whether or not memory manager returns STATUS_IMAGE_NOT_AT_BASE
 - This is returned when the PE Header's OriginalBase does not match the ASLRed (or collided) address
 - But since we are NOT a SEC_IMAGE, the memory manager does no such check
 - It always picks an available address!
- Is there a way to FORCE a collision to happen for a data section object, without having to pick a static base address and doing yet more patching?

DEMO

Conclusion

Key Takeaways

- The hotpatch API is an easy, efficient way to inject DLLs in arbitrary processes
 - Either from the kernel (no special rights required)
 - Or from user-mode (must open a handle to the process)
 - DLLs must be made “survivable” by playing reference count tricks
- Known DLLs can be redirected and hijacked to cause other DLLs to load instead of the ones the application and/or debugger/AV think are being loaded
- The loader can be tricked into believing that a non-image file, paged-file backed data section object is actually a DLL
 - But one must force it to use `PAGE_EXECUTE_READWRITE` to get code execution, unless DEP is disabled

Defense-in-depth Suggestions

- Microsoft should clean up the loader
 - Known DLLs should perhaps be validated to make sure they are in the expected directory
 - DLLs loaded through *LdrLoadDll* should be validated as being real image section objects – and failure codes should not be ignored!
- IR and forensic analysts/specialists should check contents of \KnownDlls directory to audit for any foreign known DLL
 - Use !object, dt nt!_SEGMENT and !ca to identify real file object
- Don't rely solely on debugger/kernel notifications
 - Scan VADs and other address space mappings
 - Be wary of “execute” pages associated with shared memory that is page-file backed (i.e.: not SEC_IMAGE)
 - Be wary of remote thread execution of *LdrHotpatchRoutine*

QA

- Greetz/shouts to: adc, lilhoser, msuiche, j00ru, PLA

- See you at NoSuchCon/Recon!

PS. Asiana Airlines: Still waiting for my laptop....



CROWDSTRIKE